Chapter 6

Developing Mobile GUIs

The rose and the thorn, and sorrow and gladness are linked together.

Saadi

6.1 Introduction

In Chapter 5, we saw why and how to build generic user interfaces. The two types of interfaces that dominate computing today are Graphical User Interfaces (GUIs) and Voice User Interfaces (VUIs). So, when we specialize a generic user interface, we are typically specializing it to either a GUI (of which we will consider text-only user interfaces to be a subset) or a VUI. In this chapter, we will look at GUIs, in Chapter 7 we will look at VUIs, and in Chapter 8 we will see how to build multimodal user interfaces that use multiple channels to reach the user.

Let us remember our final goal: building mobile user interfaces. Mobile user interfaces inherently have different requirements than their stationary counterparts because of the dimensions of mobility and the mobile condition of the user. The dimensions of mobility affect design and implementation of user interfaces in two fundamental ways. The first is that the user interface has to accommodate functionality that relates to the dimensions of mobility. For example, user interfaces must be available on all of those devices through which the user of an application may access a system. Second, the dimensions of mobility create various concerns that require further separation of concerns when building user interfaces. Today's state-of-the-art techniques in model-view-controller (MVC) and presentation-abstraction-control are incomplete in treating these concerns so we will first examine them and then examine enhancements and alternatives to the existing techniques that allow us to design and implement with the proper separation of concerns for the new concerns introduced by the dimensions of mobility. We have already begun this process by looking at building generic user interfaces. Generic

user interfaces simply model a user's interaction with the system (independent of the modality and the communication channels).

Because the process of building user interfaces for mobile applications is considerably more complex than their stationary counterparts, we will subsequently use UML as the tool that documents and drives the process of developing our user interfaces. We will note again that there are no specific methodologies recommended by OMG (the organization that maintains the UML standard) in regarding the use of UML to build user interfaces. Because there are no standards, we have selected some suggested techniques that fit our needs, namely development of user interfaces for mobile applications.

We will start by looking at the state-of-the-art in separation of concerns (user interface logic, business logic, publishing to multiple interfaces, delivery through multiple channels, etc.) when building user interfaces today.

6.1.1 Today's State of the Art: PAC, MVC, and Others

Before we lay out some options in building mobile user interfaces, let us look at our goal and the assumptions we make to narrow the solution set for the goal:

Our goal is to design and implement the user interfaces of our mobile applications so as to minimize the development effort and maximize the robustness of the user interfaces.

As you recall, we face multiple challenges in developing user interfaces for mobile applications. There are a multitude of devices and platforms used by the consumers (device proliferation), the user interface must be robust enough to allow the modalities that fit the condition of the mobile user at the time of using the system (support for a wide variety of user interfaces), and the user interface of one application may need to adapt itself to be used under a number of system conditions (low battery, poor QOS, and low device user interface capabilities). Because of the relative short lifetime of the mobile device acceptance in the marketplace and the large permutations of possible platforms (mobile operating system, hardware, network, deployment, etc.) we have to do our best to construct the device so that code is maintainable, extensible, and flexible. Whatever problems you may have faced in maintainability, extensibility, and flexibility of software for stationary applications are permutated by the dimensions of mobility.

Let us take a step back now. As we all know, software is somewhat like radioactive materials: It is always decaying. This decay is caused by the changing needs of the users of the system and the ever-evolving tools that serve those users. Given this decay, another genetic trait of software development is that there are always additional requirements and modifications during this decay process. So, once we have the first version of a piece of software, we must maintain it. Obviously, many of these additions and modifications are going to be additions and modifications to user interfaces or cause additions and modifications to user interfaces of the system. This problem is not unique to mobile applications. It is shared by all software applications. Although developing user interfaces for mobile applications is a problem compounded by the dimensions of mobility, developing the user interface to any software application is a fairly complex task. This complexity has given rise to several techniques that aim at easing the task of development of

the user interface. Now, remember that by development we do not just mean the initial creation of the user interface, but the creation and maintenance of the user interface over the entire life cycle of the application.

The choice of the technique we use in developing the user interface depends on the core technologies used and the architecture of the system. For example, there are a series of techniques developed for building PC-based applications and other techniques developed for building Web-based applications. In the case of mobile applications, we face two general types of user interfaces: those that use the mobile device for rendering some or all of the user interface and those that use the end device merely as a communication channel to the user. An example of the first is a networked PDA application; an example of the second is a telephone used to communicate with a VUI at the other end of the phone call. All of the techniques that we will discuss in this text for building mobile user interfaces are primarily concerned with one aspect of software development: separation of concerns. These concerns include whatever we have experienced with developing user interfaces for stationary applications (separating business logic from presentation logic, separating validation from presentation logic, etc.) and are permutated by the dimensions of mobility. In this way, our goal will be to point out techniques that allow for separation of concerns, be it the typical concerns of developing any user interface or the concerns of mobile applications, to reduce the development effort in building user interfaces and creating the best experience for the end user. Unless you are building an embedded software application for only one type of device, you will find these techniques useful. But, be forewarned that none of these techniques are the antipattern (sometimes referred to as the silver-bullet or golden-hammer antipattern). The technique that you use must fit the problem that you are trying to solve and the needs of the problem are something that you as the engineer must assess.

There are a variety of software development techniques for developing the user interface to stationary applications, but we will focus on those techniques that have evolved from the study of object-oriented programming and design patterns. If your chosen language for building your mobile application is C++, Java, or another object-oriented programming language, these techniques will apply directly. However, even if you are using a language such as C (and the relevant tool sets), the concepts will still apply. You may need to apply some creativity (or read up on writing object-oriented applications with C) in adapting these techniques to the language of your choice.

Let us start with what is probably today's most popular technique for separation of concerns when it comes to building object-oriented user interfaces: the model-view-controller technique.

Model-View-Controller

Model-View-Controller (MVC) is an object-oriented design pattern for separation of concerns of applications with user input (see Figure 6.1). MVC is best defined by Buschmann, Meunier, Rohnert, Sommerlad, and Stal (also known as the "Gang of Five") in one of the staple texts of software application development called *Pattern Oriented Software Architecture: A System of Patterns*. MVC divides an interactive

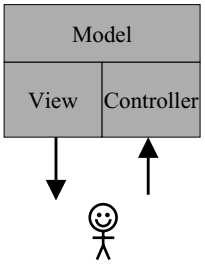

FIGURE 6.1. MVC Pattern.

application into three areas: processing, output, and input [Buschmann et al. 1996].

The model is the internal implementation of the application and does not encapsulate any data or have any behavior related to interactions with the user or the presentation of data to the user. The view encapsulates any output through the user interface to the user. What you can view on the screen or hear on the phone is rendered by the view. The controller processes the input of the user into the system. The text typed into the system, the mouse events, and the voice recorded by the system all come through the controller. The system may have one or more views and controllers. The controller allows the user to enter input. It then can modify the model. These modifications are reflected in the user interface through the view(s). MVC allows separation of three different concerns: receiving input from the user (controller), implementing components that model business logic and operations that build the core functionality of the application (model), and presenting information to the user (view).

MVC is widely implemented in stationary client applications and server-based (thin-client) Web-based applications. In such systems, there is typically only one type of view (HTML) and one type of controller (PCs and the relevant peripheries). Minor differences in things such as browser versions and monitor sizes are typically taken care of by work-arounds rather than by creating multiple views. When it comes to mobile application development, MVC has a couple of disadvantages. First, proliferation of views and controllers becomes unmanageable and very difficult to maintain as mobile applications have multiple user interfaces rendered through multiple channels and can receive input from numerous controllers. Second, the inherent asymmetry in treating the input and the output from the user to the model compounds the effect of this proliferation problem. For example, a system that offers a VUI and an HTML user interface for its users would need at least two separate controllers, one that can receives user input through a voice channel and another that receives input from the user through HTTP. Likewise, two different views would be needed, one that renders a GUI in HTML and another that renders an aural user interface through playback of audio. If we wanted access to the aural user interface through the PC as well as the telephony system, we would end up with two controllers and two views for the VUI. It is easy to see that the user interfaces and channels to be supported for a mobile application can become unmanageable. The maintenance of the controllers and views can become particularly unwieldy. The separateness of the views and controllers has

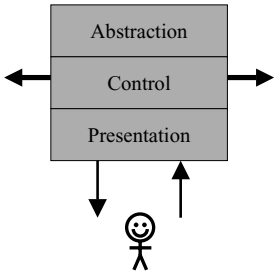

FIGURE 6.2. Presentation-Abstraction-Control.

another negative side effect: Maintaining consistency among the different views and controllers becomes cumbersome. For example, if a field has to be added to the HTML GUI, we must make sure that it is added in the same analogous point in the VUI, based on mapping the GUI interactions to the VUI interactions.

In addition to these problems, MVC does nothing to take into account the other dimensions of mobility. Namely, there is nothing that accommodates the adaptability of the controllers and views based on the dimensions of mobility such as location, QOS, device power supply, or device capabilities.

MVC still gives us some value in separating the three major concerns, but its tightly coupled and asymmetric nature, as well as its inability to treat multiple views and controller types elegantly, makes it less than ideal for user interfaces in mobile applications. Let us continue our search through existing techniques by looking at a similar design pattern, also exposed by the "Gang of Five" called PAC.

Presentation-Abstraction-Control

Presentation-Abstraction-Control (PAC) is an object-oriented design pattern that separates the concerns of a system by breaking it down into loosely coupled agents, each responsible for one task (see Figure 6.2). The Presentation-PAC architectural pattern defines a structure for interactive software systems in the form of a hierarchy of cooperating agents [Buschmann et al. 1996]. Every agent internally has components that serve one of three tasks: those components that abstract away the core functionality and data used by the agent (abstraction), those components that provide access to the agent (presentation), and those components that control the interactions between the abstraction and presentation layers (control). Note that the PAC pattern is similar to the MVC pattern in that it hides the internal implementation of the logical functions of the system from the user interface (i.e., the abstraction layer hides the business logic).

In PAC, the separation between the user interface and the functionality of the internals of the application is made by using the control component to pass messages back and forth between the two layers. Let us look at an example of how we can apply PAC in Figure 6.3.

Let us say we need to build a reusable user interface component that collects billing information from a user when he or she is purchasing something online using an HTML-based browser. This component is probably a panel that has some buttons, labels, and text fields. Internal to the system, this information may be

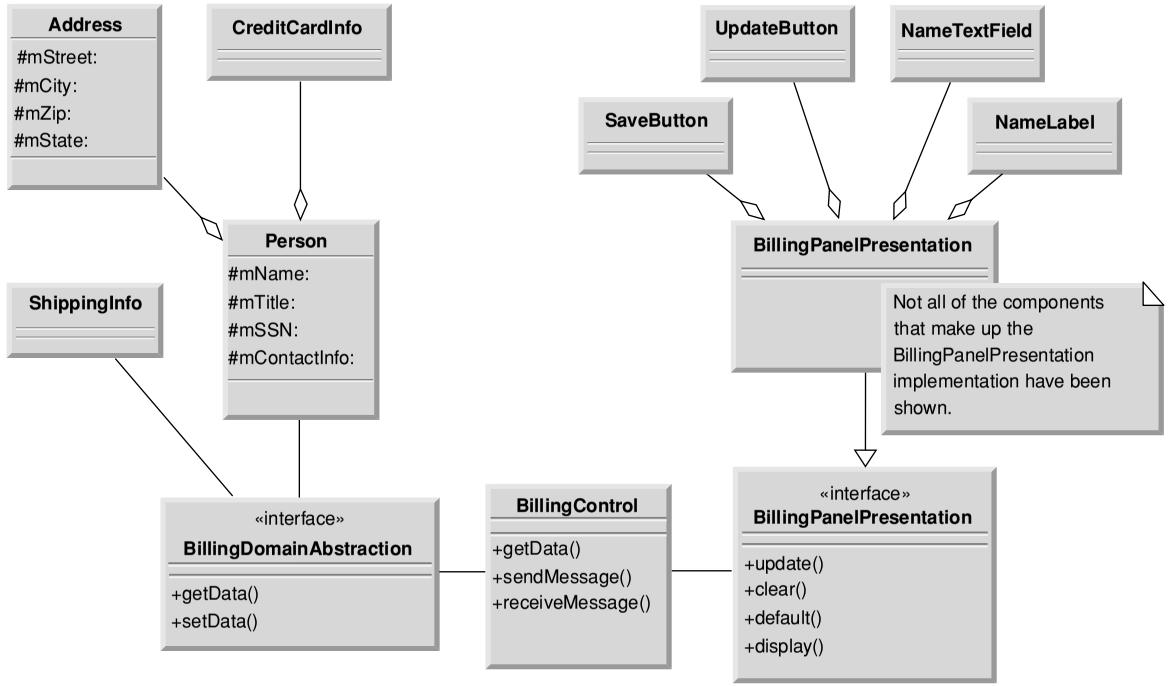

FIGURE 6.3. UML Class Diagram of a Sample PAC Implementation.

encapsulated in several different objects. We create an abstraction (whose interface is seen in the UML class diagram) to get the appropriate data out of the domain. The implementation of this abstraction may exist in the domain model or we may need to implement it (depending on whether the existence of the relevant information grouped as billing information is necessary or not). The abstraction provides the necessary behavior to exchange data with the BillingControl class. The implementation of the interface between these two components (the abstraction and control parts of PAC) is determined by what the controller needs from the abstraction. The presentation is the panel itself, probably dynamically generated using a scripting language such as JSP or ASP.

One key thing to note here is that the components of PAC are very decoupled. Although the example that we have shown is a low-level one, PAC scales very well. Various components can be tied together in a very decoupled way as it is very legal for controllers to communicate and collaborate with one another. Consequently, making complicated user interfaces based on simple components is more natural to PAC than it is to MVC because composition of the agents can be done without violating encapsulation. In fact, as defined by Buschmann et al. earlier, it is this treelike hierarchy of agents that define the PAC pattern.

In this way, we can imagine that because of its flexibility to composition and delegation and because of its decoupled nature, it is even possible to scale up PAC so that the various parts of PAC are completely separate processes. These properties are precisely what make PAC a good fit for mobile application development.

The PAC pattern fits the problem of mobile user interfaces much better than MVC. First, it provides us a well-defined place to hook in the various infrastructure pieces that take care of the dimensions of mobility and affect the user interface without exposing this functionality to the core logic of the application: the control component. This means that the control component can communicate with the location sensitivity system, the voice recognition engine, the speech synthesis engine, and all the other subsystems that we need to use to control and produce our user interface without violating the separation of concerns among the abstraction, shielding the business logic, and presentation.

PAC also provides one single layer for presentation, allowing us to encapsulate the channels and modalities of the presentation in the same layer. PAC gets us closer to what we need than MVC, but it still does not directly address all of our needs. We need to embellish on it to get an approach that will fit mobile applications well.

Transformation-Based Techniques for Mobile Applications

As we reviewed in Chapter 1, the first versions of mobile applications were essentially either custom embedded applications with custom architectures and designs or fully centralized applications with proprietary devices and networks as the end nodes. The first attempts at building mobile user interfaces has been to transfer today's HTML-driven Web model to handheld mobile devices. WAP's WML (which we will review in detail later in this chapter) and NTT Dococo I-Mode's cHTML are examples of subsets of HTML functionality. The goal of such markup languages is to take a subset of functionality of HTML. This has two benefits: 1. Only a subset

is needed for devices with limited capabilities, bandwidth, power supply, etc. and 2. having a subset enables us to have a simpler and smaller browser that uses less of these scarce resources.

Because Web content is mostly in HTML, this means that HTML has to be transformed to the markup language supported by the target device. But because there are a variety of devices and slightly different implementations and variations of the markup languages, developers were left with a significant problem: how to automate the task of publishing to these various user interfaces. As we mentioned previously, one way is to continue using MVC and create multiple controllers and views. This is really not practical because as the number of controllers and views grows, maintenance becomes unmanageable. The PAC pattern gives us a better approach because there is only one presentation component that interacts with the user.

Developers began using two techniques to complement both PAC and MVC:

1. *Transcoding*: If the content is initially in HTML, we are dealing with a "view" of the existing system. Transcoding techniques focus on extracting the information out of this view to create an intermediate format that can in turn be used to produce other views. The process of creating this intermediate format is referred to as "transcoding." Prior to use in the moble context, the term transcoding typically meant conversion of one compressed format to another. And, as in the case of conversion of one compressed format to another, there is almost always some loss of data in conversion of HTML (or another markup language) into the intermediate format. The intermediate format is used like a generic user interface and then transformed to the various views using XSL or a similar technology.
2. *Transforming*: Although we can start with HTML (or some other presentational view of the system) and convert to other views of the system, this is a solution that should be done only as a last measure. The preferred situation is that all content is initially produced in XML that gives a presentation-neutral view of the system. This content can then be transformed to the appropriate views using XSL or similar technologies.

There are two main differences between transcoding and transforming. First, in transcoding, we are starting out with some final (specialized) content, not raw (generic) content. Second, transcoding is typically lossy and needs special instructions whereas transforming is not lossy and should not need special instructions (other than the transformation). Both transcoding and transforming are complementary to MVC and PAC. Figure 6.4 shows how the content produced by a system using PAC is transcoded and transformed to other forms of content for access by multiple user interfaces.

Note that transcoding and transforming the output of the system only solves the problem of publishing multiple types of output to the user. It does not deal with the fact that the input from the user may be coming from a variety of disparate channels such as HTTP, WAP, VoIP, or POTS. This problem has typically been solved by using a proxy that resides, on the server side, between the Web-based system that

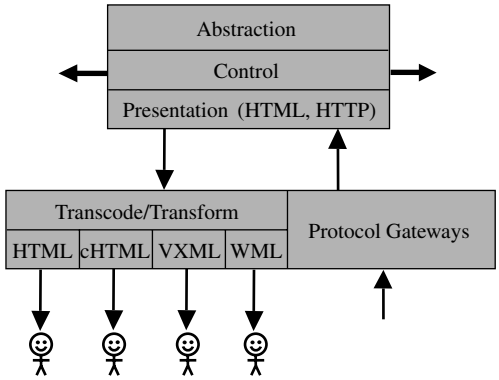

FIGURE 6.4. Using Transcoding/Transforming Techniques to Complement PAC in Producing User Interfaces for Mobile Applications.

supports HTTP and the infrastructure and communication protocol native to each type of mobile device. For example, in the case of WAP, WAP gateways act as a proxy and a protocol converter to convert all of the user input sent from the device to the WAP gateway in the native WAP protocol implementation to HTTP.

Figure 6.5 shows how generic XML content can be produced by the presentation layer and transformed using XSL (or any equivalent transformation technology can be used) to the final markup language to be used by the device. Once transcoding and transforming solutions were deployed, it became obvious that there needed to be a solution for an intermediate user interface format, one that treats the interactions of the user with the system in a generic way and independent of the properties of the specific devices. This, in turn has given rise to the genesis of several efforts including XForms, which we looked at in Chapter 5, and User Interface Markup Language (UIML).

We will look at UIML later in this chapter. At this point we should take a step back and note that UIML and XForms take fundamentally different approaches in allowing developers to create a generic user interface. As we saw in Chapter 5, XForms defines distinct and discrete controls and elements that define a language for building a generic user interface; XForms is an XML application. In contrast,

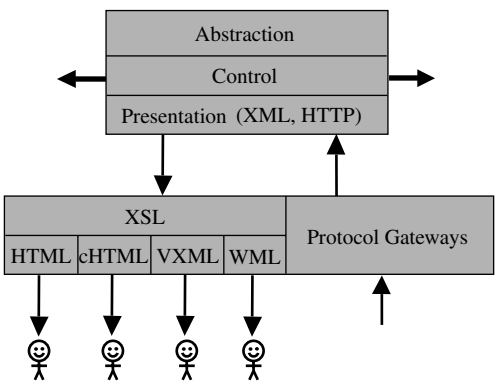

FIGURE 6.5. Using Transcoding/Transforming Techniques to Complement PAC in Producing User Interfaces for Mobile Applications.

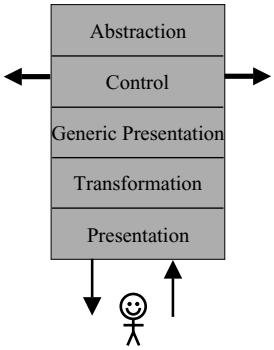

FIGURE 6.6. PAC-TG: A Variation on PAC for Mobile User Interfaces.

UIML is an XML-based vocabulary intended to define other XML-based applications that describe user interface interactions; UIML is not an XML application; rather, it is an XML vocabulary similar to XML Schema. UIML, in a way, is a metalanguage intended to create other languages that are used to build user interfaces. Whereas UIML itself can be used to define XML applications to generate generic user interfaces such as XForms, it can also be used as a "metageneric" user interface in that it can be used by developers to define XML applications suitable for various types of user interfaces.

Note that all of the techniques discussed so far focus on selecting the right high-level approach in building our user interfaces. To build mobile user interfaces, we will combine the best of what these techniques have to offer.

6.1.2 PAC-TG

In this text, we will recognize PAC-TG, short for Transformation of Generic Presentation-Abstraction-Control, as a high-level design pattern for creating user interfaces to mobile applications. This pattern is not a new pattern (because patterns, by definition, are not invented but rather are recognized by prevalence of use and benefits). It is merely a specialization of the existing PAC pattern as recognized by Buschmann et al. Figure 6.6 shows how PAC-TG builds on PAC.

When we looked at the PAC pattern, we saw that it breaks down the task of creating a system or subsystem, in our case the user interface, into a series of agents. Each agent has three different components of abstraction which gave us an interface to the data and behavior model of the system; the presentation, which gave us the final mechanism to render the interface; and the controller, which controlled the interactions between the presentation and the abstraction.

We are going to specialize this pattern by making a restriction and an addition. First, we are going to restrict the presentation components to encapsulate information and behavior about interactions with the user that are independent of the final user interface viewed by the user. We discussed generic user interfaces in Chapter 5. Then, we are going to add transformation components that specialize the generic presentations. The generic presentation and transformation may be implemented in several different ways. We have already discussed some aspects of generic user interfaces and will discuss them further. We will also look at some implementation examples for the transformation.

Now, as we would with any other recognized software pattern, let us define the intent, motivation, known uses, business domains, problem forces, benefits, and liabilities. Then, we will delve into some sample implementations of PAC-TG.

Intent

The intent of PAC-TG is to combine PAC agents and transformation techniques to structure the production of multiple user interface types to a common application for various devices. Such subdivision separates the concerns of functional implementation of the application, interactions with the user through the user interface, and the variations in the user interface types presented to the user.

Motivation

PAC-TG is a modification on PAC that uses the treelike hierarchy of PAC, inversion of control, and the concept of specialization of generic user interfaces to various specialized user interfaces. Every PAC-TG agent is composed of at least five components, one component that provides an abstraction to the core functionality of the application (abstraction), one component that provides a generic user interface to be used by other components or systems (generic presentation), one or more components that transform the generic presentation to a specific presentation or presentations, one or more components that produce final user interfaces with which the users interact (presentation), and one component that facilitates messaging among all of the other components (control).

Known Uses

Commercial publishing frameworks and transcoding products include IBM's Transcoding Publisher, IBM VXML Portlets, and open-source projects such as Apache's Cocoon. (Cocoon components can be arranged both as an implementation of MVC or as an implementation of PAC depending on the usage as Cocoon as a component framework.) MATIS also uses a roughly equivalent pattern called PAC-Amodeus (see the last subsection in Section 6.1.2). Nunes' Wisdom architecture and methodology [Nunes 2001] also outlines use of this pattern without specific recognition of it.

Related Patterns

As we have mentioned, this is a variant on the PAC pattern. Various other low-level patterns such as the Visitor and Façade patterns can be used in the internal implementation of individual components of a given agent.

Business Domain

Development of applications that require more than one rendition of the same user interface or multiple types of user interfaces fall into the business domain.

Problem Forces

One of the problems to consider while implementing this pattern is that this is a high-level design pattern. The internal implementation of this design pattern can vary greatly. As we mentioned in the case of PAC, because of the loosely coupled

nature of PAC-TG, agents and/or components can be run within separate processes. The method by which they communicate (protocol, etc.) is not restricted (i.e., they could be native protocols such as RMI and COM or open system protocols such as HTTP and CORBA).

Also, note that every agent (package of presentation, abstraction, control, and transformation components) maintains its own state. Because the user may give the system input while state information is being exchanged within the different agents, transactional integrity must be provided to make sure that illegal states are not possible.

Lastly, PAC-TG treats the concern of creating multiple user interfaces but does not treat the fact that these multiple user interfaces may be using multiple channels to reach the user (at the end device—for example, a VoIP voice channel as opposed to a regular POTS-based voice channel).

Benefits

We can outline the following benefits in using the PAC-TG pattern:

1. *Separation of concerns between the internal implementation of the business application and the implementation of the user interface.* This separation of concerns allows for possible reuse of components (though reuse takes more thought than merely utilizing design patterns), better scalability by distribution of the model and the interface concerns over different processes, and easier code maintenance during the application life cycle. This benefit is inherited from *PAC*.
2. *Separation of concerns between the device and interface-specific interactions of the user with the system and the different generic methods by which the user can affect the state and behavior of the system.* This separation allows us to develop reusable transformation components that transform a particular set of generic interactions to one or more specialized user interfaces. It also provides us with a tool to avoid very fast growth of the development effort to build n user interfaces for m types of device accessing a single application with which the interactions of various devices are fairly alike.

Liabilities

The first and biggest liability of PAC-TG is possible performance degradation because of the additional layers of abstraction (of course, this depends on the implementation). This is due to the higher number of objects instantiated and managed if implementation is object oriented. So creating each user interface in a custom way will invariably yield a user interface that requires less computing resources. This performance problem is more visible when all of the components of PAC-TG are being executed on the same process and the same physical device. To alleviate performance bottlenecks, distribution of the different components is recommended because the loosely coupled nature of PAC-TG allows this.

The second liability is that, as in the case of PAC, PAC-TG is a complex pattern to implement. Because of the various ways that it can be implemented, it typically requires considerable experience in recognizing the appropriate behavior

of the interfaces and boundaries between the agents and between the components within the individual agents.

Examples

We can implement PAC-TG in three ways. In the first type, the control component may facilitate communication among all of the other layers. This is seen in Figure 6.7, where we have shown a Type 1 PAC-TG implementation for the billing panel that we discussed in the previous section.

This is the simplest implementation of PAC-TG. Note that we have shown the specific presentations in the model as multiple classes whose code is generated by the framework and not written by the developer. This is not the best implementation of PAC-TG as it creates a high level of coupling between the control component (called PACTGBillingControl in our example) and the specialized presentations of the user interface. The single control component in this implementation is responsible for communication among all of the other components.

Alternatively, we can break the control components into two separate control components: one that facilitates control and communication between the generic user interface and the abstraction of the system and the other that facilitates control and communication among the generic user interface, the transformation components, and the final user interfaces produced. Figure 6.8 shows how our Type 1 implementation shown in Figure 6.7 can be modified to do this.

The advantage in using Type 2 PAC-TG is that the type of behavior required to facilitate control and communication between the generic user interface and abstraction and those required to facilitate control and communication among the transformers, the generic user interface, and the specialized user interfaces are fundamentally different. So, by separating these tasks, we achieve a good separation of concerns.

We can make this yet more efficient by using Type 3 PAC-TG (Figure 6.9), where the control component is broken into two separate components as in Type 2, except that it communicates with the generic presentation layer instead of the control component that allows for communication between the abstraction and generic user interface. Once again, this provides us with a couple of significant improvements. First, there is less indirect communication. In Type 2, data going from the generic presentation to the specialized presentation layer have to go through at least two layers. In this model, we reduce that to one layer. This improves efficiency. Second, this setup is more in keeping with the spirit of the design pattern in enabling a high degree of decoupling so that each set of agents, containing abstraction, control, generic presentation, transformation, and specialized presentation components, can be run in a very decoupled manner (separate thread, separate process, or even possibly a separate operating environment altogether).

This brings us to the end of our discussions about separating the concern of creating a generic user interface and a specialized one. Our next task is to see this applied in building GUIs for mobile applications.

As we noted before, this pattern does not do anything to take into account the concerns of dimensions of mobility. Particularly, multichannel communication, location sensitivity, resource constraints of the device, and QOS conditions are

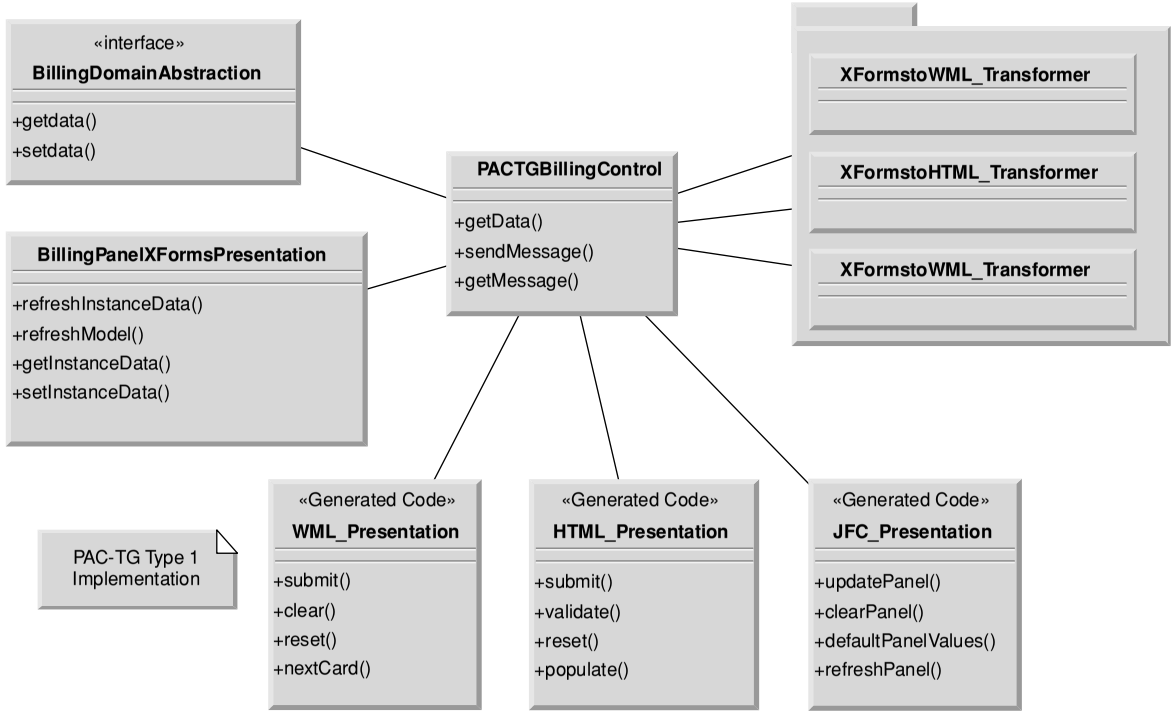

FIGURE 6.7. PAC-TG Implementation Type 1.

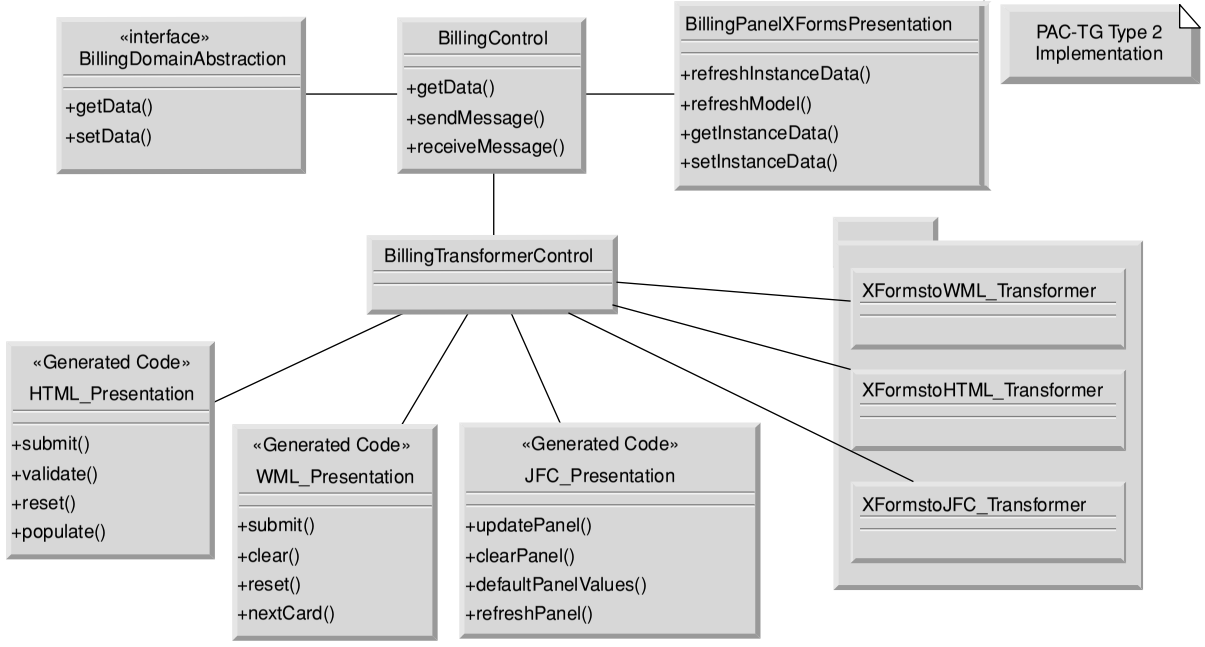

FIGURE 6.8. Type 2 PAC-TG Implementation.

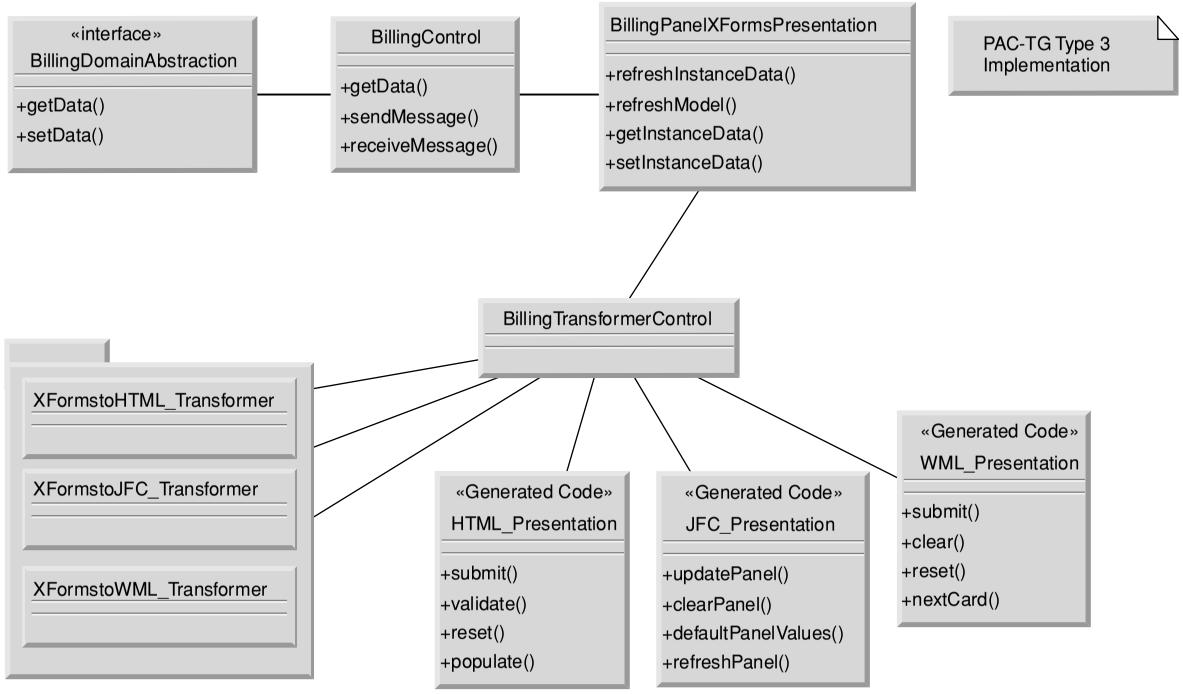

FIGURE 6.9. PAC-TG Type 3 Implementation.

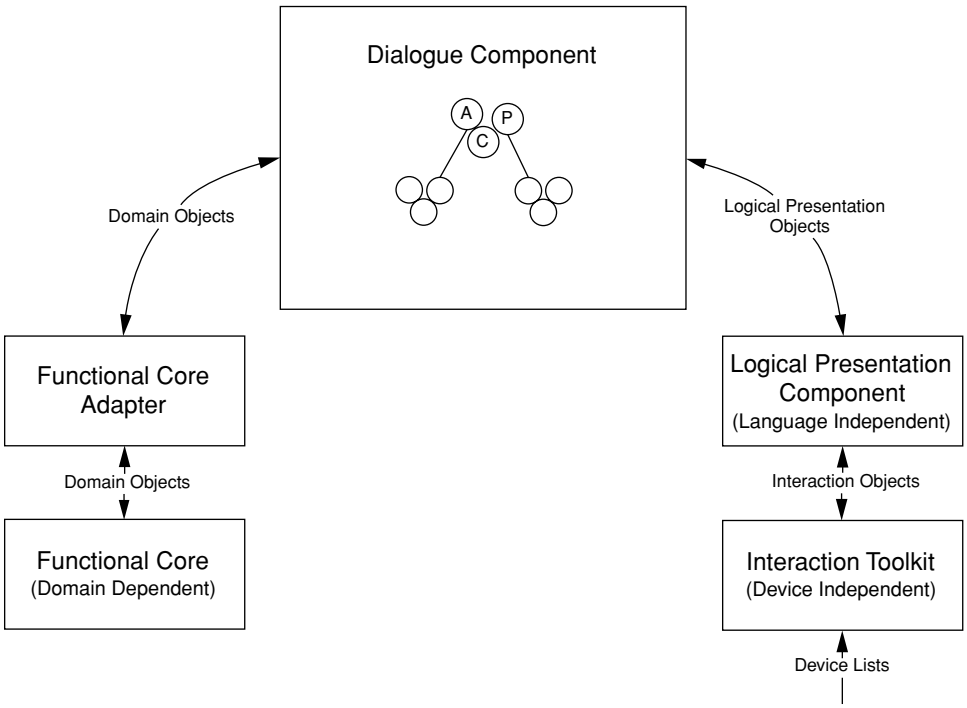

FIGURE 6.10. The PAC-Amodeus Functional Components [Coutaz 2002].

not taken into account. Remember that for a design pattern to be recognized, it must be applied and discovered rather than invented. Because mobile application development is a less mature software development field, there are no current patterns used among software developers to treat the dimensions of mobility.

It is, however, intuitive that we could extend PAC-TG to treat dimensions of mobility through creating additional types of control components that connect to tertiary components treating the various dimensions of mobility. We will leave this implementation to the reader of this text and hope that such patterns become recognized by the industry and are ripe for introduction in the next edition of this text.

Now, let us look at building some simple single-channel GUI applications for mobile applications.

PAC-Amodeus

Introduced by Coutaz [Coutaz 2002], PAC-Amodeus is very similar to PAC-TG, but it attacks the problem differently. The functional components of PAC-Amodeus are shown in Figure 6.10.

Of particular interest to us is the Interaction Toolkit Component, which provides device independence. This component represents a set of agents that provide functionality such as transformation needed for specialization of a generic user interface to the final user interface to be rendered for each particular device, modality, etc. The Dialogue Component encapsulates the functionality previously modeled by the PAC pattern and the Logical Presentation Component presents us with the

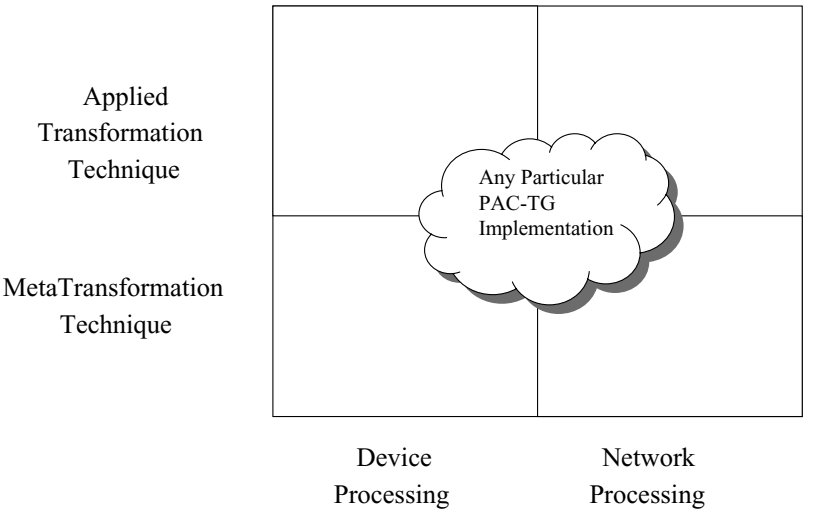

FIGURE 6.11. Division of PAC-TG Implementation Techniques.

generic user interface layer that provides a layer where the interactions of the user with the system are modeled in a user-interface-generic manner.

Essentially, PAC-Amodeus introduced by Coutaz covers PAC-TG at a high level and adds abstraction layers for the business logic (Functional Core Adaptor and Functional Core) that separate the access the Dialogue Component needs to the engine that models the logic from the access interface itself.

Coutaz than introduces MATIS (Multimodal Airline Travel Information System), which allows a user to retrieve information about flight schedules using speech, direct manipulation, keyboard, and mouse, or a combination of these techniques [Coutaz 2002]. MATIS is then used as an example of a PAC-Amodeus-based system. The referenced work by Coutaz is recommended reading to become more familiar with the details of this pattern.

6.1.3 Single Channel Specialization of Generic User Interfaces to Graphical User Interfaces

In this chapter, our focus is in understanding the implementation of GUIs for mobile applications. When dealing with GUI applications, we typically have a single channel of communication between the device and the network. We will consider multichannel user interfaces in Chapter 8. In the previous section we discussed PAC-TG as a design pattern that can help us produce multiple GUI-based user interfaces for a single functional core application. In this section, we will focus on the various implementation methods for the PAC-TG design pattern. The techniques used for such implementations fall somewhere on the plane graphed in Figure 6.11.

As shown in the picture, any PAC-TG implementation technique can distribute the processing between the end device used by the user as the interface to the system and the other processing units on the network (peers, servers, etc.). Also, every implementation technique may use a well-defined language and tool set for defining the generic user interfaces and transforming them (such as XForms for

implementing a generically defined interface and XSL for transforming the XForms documents to specific markup languages). Alternatively, it may use a metalanguage and relevant tools such as UIML, which we will look at in this chapter as a tool to define metarules that can be used, at run time or batch time, to generate generic user interfaces and the relevant transformations.

We have already implemented XForms as a well-defined application of XML that allows us to create user interfaces whose interactions with users are independent of the device type. Using XSL is also a very popular method of transforming XML content to other markup languages such as WML, VXML, and HTML. Using XForms and XSL as the implementation tools for PAC-TG would put us on the top half of the plane of Figure 6.11.

If the core of the application (exposed to PAC-TG through the abstraction) resides on the network, then it only makes sense that the generic user interface is produced by the network (servers, peers, etc.). (Once again we refer to any processing being done on anything but the client as processing being done on the network because we are trying to treat the problem in an architecturally independent way.) If the core of the application (exposed to PAC-TG through the abstraction) resides on the end user device, then the production of the generic user interface and its transformation are both performed on the device itself. It is crucial that whatever tool is selected to create the generic user interface and perform the transformations is flexible enough to be used on the end device as well as on the servers and peers on the network. XForms and XSLT technologies, respectively for the production of the generic user interfaces and the transformations, provide us with such flexibility. We can have XForms browsers that reside on the device itself and transform the XForms controls and interactions to the appropriate user interface for the end device or we can have the transformation of the XForms document happen somewhere on the network and send simple markup languages such as WML, XHTML, HTML, or VXML to the browser. Note that this is as if PAC-TG is fully implemented outside of the end-user device (servers or other peers), we are practically looking at a model where either the device is a "dumb" client (such as a regular old telephone) or it has a browser such as a WML or HTML browser that simply converts the final markup language to the look-and-feel made available on the device.

Techniques that define the infrastructure for defining generic user interfaces and transformations thereof (metatransformation techniques) are only slightly different from their applied counterparts. This is because, by definition, generic user interfaces are about defining the metainteractions of the user with the system as opposed to the exact interactions. So, XForms and similar tools are in a way metatools. However, UIML and similar techniques can be used to also define tools such as XForms. There is a benefit and a loss in this case. Obviously, the metatool, something like UIML, is more flexible, but it is also more ambiguous and requires more decisions to be made by the designers of a particular application, more custom code, and, therefore, more complexity and less reliability in the final system. However, there is one very big advantage that a tool such as AUIML (Abstract User Interface Markup Language) provides that we have not mentioned:

SIDE DISCUSSION 6.1

The Common Thread in Generic User Interfaces

In a document aptly titled "Towards Convergence of WML, XHTML, and other W3C Technologies" by Dave Ragget and Ted Wugofski, they mention the common threads among the various tools to facilitate the creation of generic user interfaces and provide a transform mechanism [Ragett and Wugofski 2000]:

- extensible event handling mechanism,
- a means of providing default event handlers (templates) and overrides,
- a means of navigating to another dialogue or document in response to any event, and
- a means of managing state information in response to an event.

Keep these in mind when you look at the various tools that we introduce throughout this text. Note that the decoupled nature of PAC-TG allows for a particularly natural implementation of the last two (dialogue navigation and state management).

Metatools map well to UML. And because of this, they offer us the only possibility of defining a fully automated user interface generation system from UML to date.

We will look at UIML later in this chapter and subsequently see how it differs from XForms as a tool for generation of generic user interfaces.

Let us look next at how we can build a GUI for mobile applications.

6.1.4 GUI Specialization on the Server

To display a GUI to the user, we need to specialize the generic user interface to what the user eventually sees. The simplest way of doing this is to specialize the user interface on some server:

1. *Thin-Client Markup Language–Based Applications*: These are the run-of-the-mill WML, VXML, XHTML, HTML, or other types of markup languages that can be used in creating static documents or produced dynamically and then browsed on the client. WML is the most pervasive of these solutions for mobile environments to date. The generic user interface content may be transformed to the desired markup language using XSL or some other transformation mechanism.
2. *Mobile Agents*: We will take a closer look at mobile agents in Chapter 9. Although weak and strong mobility allow us to build applications that migrate to the device and do their work there (rendering of the user interface and interacting with the user), we can either use servers to select the right type of agent to be delivered to a device or we can have the server produce an agent and send it to the server in an automated fashion. An example of the first is a server that provides a provisioning system for J2ME midlets and BREW applications at the same time. The second is a bit far-fetched with the technologies that are available today; nevertheless, it is a possibility.